SIAM J. SCI. COMPUT.
Vol. 35, No. 3, pp. A1669–A1688

AN EFFICIENT SEARCH ALGORITHM FOR MINIMUM COVERING POLYGONS ON THE SPHERE*

NING WANG†

For my mother, in loving memory

Abstract. One of the computationally intensive tasks in the numerical simulation of dynamic systems discretized on an unstructured grid over the sphere is to find a number of spherical minimum covering polygons of given locations, whose vertices are chosen from the grid points. Algorithms have been proposed attempting to perform this task efficiently. However, these algorithms only reduce the linear search time for each polygon vertex candidate by a constant factor, and their polygon search algorithms are mostly heuristic and tailored for specific classes of grids. With the increase in grid resolution, the number of unstructured grid types, and dynamic generation of variable resolution grids, these algorithms are no longer suitable for the computational task. It is necessary to develop a more general, efficient, and robust search algorithm. We propose an algorithm, built on a modified kd-tree algorithm, to search for minimum covering polygons of given locations from a set of grid points on the sphere. After an $O(n \log n)$ time initialization to construct the kd-tree from n grid points, the proposed algorithm takes an $O(\log n)$ expected time to find the minimum covering polygon for a given location on the sphere (or the same expected asymptotic time with a smaller constant to obtain an approximate solution). We present the modified kd-tree algorithm, showing its applicability to the search problem. We present the search algorithm for the spherical minimum covering polygon and an analysis of the algorithm's computational complexity. To demonstrate the computational efficiency of the proposed search algorithm, we apply it to the data sets of randomly generated data points on the sphere from various distributions and to the data sets of two types of spherical grids, icosahedral grid and Gaussian grid, which are widely used in the numerical simulation on spherical bodies.

Key words. minimum covering spherical polygon, nearest neighbor search, modified kd-tree, unstructured grid

AMS subject classifications. 68P05, 68P10, 65M50

DOI. 10.1137/120880331

1. Introduction.

Unstructured grids on the sphere have become more popular in recent years in the development of global models that numerically simulate the fluid dynamics on Earth and other spherical bodies [17, 22, 19, 20]. During the preprocessing (initial condition generation, data assimilation, stencil indexing computation) and postprocessing (remapping to various grids for diagnosis and visualization), various interpolation and projection operations are often required. One of the computationally intensive tasks associated with these operations is to find a set of spherical polygons from the discretization grids for a number of given locations, such that each polygon covers a given location and has the minimum combined distance from its vertices to the given location. We refer to this spherical polygon as minimum covering spherical polygon, or minimum covering polygon (MCP). We will give the formal definition of MCP in a later section. Intuitive and straightforward algorithms to search for an MCP have been suggested and implemented. These algorithms reduce the search time by presorting and indexing the grid points and latitudinal zones and by limiting the range of the geographic coordinate values for the search (e.g., [14]). However, these methods only improve the computational efficiency by a constant factor over

*Submitted to the journal's Methods and Algorithms for Scientific Computing section June 11, 2012; accepted for publication (in revised form) March 11, 2013; published electronically June 25, 2013.

http://www.siam.org/journals/sisc/35-3/88033.html

†Cooperative Institute for Research in the Atmosphere, Colorado State University, and NOAA Earth System Research Laboratory, Boulder, CO 80305 (ning.wang@noaa.gov).

an exhaustive linear search, in finding the MCP vertex candidates, and their MCP search algorithms are mostly heuristic and often designed for specific classes of grids. With the increase in the grid resolution, the number of unstructured grid types, and dynamic generation of variable resolution grids, the need has arisen to create a more general, efficient, and robust algorithm for this computational task. As one would expect, the desired MCP search algorithm should be built on an efficient proximity search algorithm on the sphere.

Proximity search on the sphere is in essence an optimization problem in metric space. Let n be the number of points distributed over the sphere. Straightforward nonhierarchical linear search methods have an $O(n)$ time complexity with or without precomputed and indexed search regions. Research in computational geometry offers a rich library of algorithms for proximity search. Nearest neighbor search algorithms for general metric space are asymptotically better [11, 25, 8] than linear time search algorithms. Furthermore, it is straightforward to show that a nearest neighbor search on the sphere with its distance metric defined as the great circle distance is equivalent to a nearest neighbor search in its corresponding ambient three-dimensional Euclidean space. Therefore, we can adopt one of those more efficient nearest neighbor search algorithms in Euclidean space [12, 6, 9, 2].

From the perspective of computational complexity, there are two classes of nearest neighbor search algorithms for Euclidean space, one offers logarithmic or sub-linear worst case query bounds at the expense of greater preprocessing time and space, and the other offers logarithmic expected query time bounds with mostly linear preprocessing space. In a d dimensional Euclidean space, for the former class of algorithms, the worst case query time are typically bounded by $O(2^{O(d)}\log(n))$ (or $O(d^c\log(n)), c \geq 5$) with $O(n^{O(d)})$ preprocessing space [1]. More recent research of the subject has been focused on the approximate algorithms for high dimensional space that alleviate the "curse of dimension," the exponential growth of query time and preprocessing space with dimension d ([3, 10, 13], for example).

From the requirement of the MCP search algorithm and practical considerations, we propose to use the kd-tree algorithm [7, 12], with some additions and modifications, in the MCP search algorithm. A kd-tree is a balanced binary tree that partitions a general multidimensional Euclidean space. The nearest neighbor search algorithm based on the kd-tree is known to be efficient in a low dimensional Euclidean space. After an $O(n\log n)$ time initialization, a kd-tree search algorithm takes an $O(\log n)$ expected time to answer the nearest neighbor query, assuming that distribution of data points are largely uniform [12]. Although grid points on the sphere are not uniformly distributed in three-dimensional Euclidean space, they are, nevertheless, rather regularly distributed in its subspace, a two-dimensional manifold (a two-dimensional sphere). Besides, spherical grid points reside in a low dimensional space, which is most important for the performance of the nearest neighbor search in Euclidean space. Numerical experiments we conducted have shown that for most spherical grids, the kd-tree search algorithm (with the required modifications) performs extremely well.

It is difficult and inefficient to use the classic kd-tree in the MCP search algorithm. The main issue is that if we have a large number of neighboring grid points that reside on the same half space, relative to the query point (a given location for an MCP search), the search of kd-tree will return many nearest neighbor points that cannot make a spherical polygon that covers the query point. Therefore, it is essential to modify the kd-tree so that the search space is limited to a specified angular area (a lune on the sphere). In other words, we modify kd-tree to do a "directional" search for the nearest neighbor.

With the proposed modified kd-tree, we can construct an efficient MCP search algorithm. Under the guidance of a few geometric propositions presented in this article, we create a search algorithm that finds the MCP for a given point with a small constant number of calls to the nearest neighbor search routine of the modified kd-tree algorithm. Briefly, for a given point p, we iteratively search the nearest neighbors of p in specified lunes and find and update the candidates for vertices of the MCP. In an expected constant number of iterations, the search algorithm goes through all possible combinations and finds the solution, a sequence of vertices that forms the MCP for p.

The article is organized as follows: section 2 discusses the feasibility of using the modified kd-tree for the MCP search problem and gives a detailed description of the modified kd-tree algorithm. In section 3, we present the MCP search algorithm based on the modified kd-tree and show its space and time complexity for the average and the worst case. Section 4 presents several experiment results to confirm the time complexity of the modified kd-tree algorithm and the MCP search algorithm. Both search algorithms are tested with random data sets of different resolutions and distributions, and the MCP search algorithm is applied to two specific spherical grids, the icosahedral grid and Gaussian grid, of different resolutions. Finally in section 5, we conclude the article with a summary of the performance, extensibility, and other possible applications of the proposed algorithms.

2. Nearest neighbor search on the sphere and the modified kd-tree algorithm. The algorithms for a nearest neighbor search in a Euclidean space are much more efficient than those in a general metric space, especially in relatively low dimensional settings. In the following, we claim that a nearest neighbor search on a sphere using the angular distance metric can be done with a nearest neighbor search in its corresponding Euclidean space.

2.1. Metric spaces of equivalent proximity relations. The next proposition and the arguments followed show that Euclidean space and angular distance metric space on the sphere are equivalent in terms of proximity relation.

PROPOSITION 2.1. *Let (X, d_1) and (X, d_2) be two metric spaces, where X is a set and d_1, d_2 are metrics on X. The two spaces have the equivalent proximity relation if for any $u_0 \in X$, two closed metric balls*

$$B(u_0, r_1, d_1) = \{x \in X | d_1(u_0, x) \le r_1\} \text{ and}$$
$$B(u_0, r_2, d_2) = \{x \in X | d_2(u_0, x) \le r_2\}$$

are equal for any r_1 and a corresponding r_2.

It is trivial to show the proposition. Since the metric balls $B(u_0, r_1, d_1)$ and $B(u_0, r_2, d_2)$ are identical when r_1 and r_2 are set to the respective nearest neighbor distances, two metric spaces must have the same nearest neighbor for any point u_0.

Now we consider two metric spaces (X, d_a) and (X, d_e), where X is a point set on the unit $(d-1)$-sphere in R^d, and d_a and d_e are the angular and Euclidean distance metrics, respectively. For $u, v \in X$, $u = (u_1, u_2, \ldots, u_d), v = (v_1, v_2, \ldots, v_d)$,

$$d_a(u, v) = \cos^{-1}(u_1 v_1 + u_2 v_2, + \cdots + u_d v_d),$$
$$d_e(u, v) = ((u_1 - v_1)^2 + (u_2 - v_2)^2 + \cdots + (u_d - v_d)^2)^{1/2}.$$

It is straightforward to verify that the two metric spaces have the equivalent proximity relation. Let θ be the angular distance between u and v. Since

$$d_a(u, v) = \theta,\ d_e(u, v) = (2 - 2\cos(\theta))^{1/2} = 2\sin(\theta/2),\ 0 \le \theta \le \pi, \tag{2.1}$$

we have $B(u_0, r_a, d_a) = B(u_0, r_e, d_e)$ when $r_e = \sin(\theta/2)(\theta/2)^{-1} r_a$.

Therefore, nearest neighbor search on the 2-sphere with respect to the angular distance metric is equivalent to the nearest neighbor search in its ambient three-dimensional Euclidean space.

Note that the above conclusion is valid for a general $(d-1)$-unit sphere in d dimensional Euclidean space.

2.2. The kd-tree algorithm. A kd-tree is a binary tree that uses orthogonal hyperplanes to partition a multidimensional Euclidean space. It is known to be an effective data structure for various geometric queries.

In the following, we give a brief description of the kd-tree and its nearest neighbor search algorithm to provide some background information for the description of the modified kd-tree algorithm in the next subsection. For more details of the analysis and implementation of the kd-tree algorithm, interested readers are referred to the cited references.

In a kd-tree, each node represents a bounded space of d dimensions, and each child node represents a subspace of its parent node. The union of all bounded spaces at any tree level represents the whole search space. The algorithm to construct a kd-tree is as follows: Starting at the root node of the tree, the algorithm selects a dimension according to a given dimension-selection algorithm and divides the space into two subspaces such that each subspace has an equal number of points. Two child nodes are then created for the two subspaces and are linked to their parent. The above procedure is carried out recursively on the subspaces until the number of points in each node at the bottom level of the tree reaches a specified number (bucket size). The nodes at the bottom level of the tree are called leaf nodes, or buckets.

The search algorithm works as follows: First, it traverses down the kd-tree recursively, following the subspaces that are on the same sides of the hyperplanes as the query point, to the leaf node. In the leaf node, it computes the distances between the query point and all data points within the node to find the current nearest neighbor point (CNNP) and the current nearest neighbor ball (CNNB) which is centered at the query point with the current nearest neighbor distance as the radius.

The search algorithm then starts a procedure to unwind the recursive search and update the CNNP and CNNB if necessary. The procedure checks if the CNNB intersects the subspace of the sibling node. If it does, the algorithm searches the sibling node by invoking this entire search algorithm at the sibling node, which includes traversing down from there to a leaf node, and possibly updating the CNNB and CNNP. If it does not, or at the completion of the search of the sibling node, the search algorithm returns (moves up) to its parent node and repeats the procedure at that level. The search terminates when it returns from the two recursive calls at the root node.

The time required to construct the kd-tree is determined by the depth (or equivalently the bucket size) of the kd-tree. It is bounded by $O(n\log(n))$ if an $O(n)$ median search algorithm is used. The space requirement for the kd-tree data structure is determined by the number of nodes in the kd-tree, which has the maximum value of $2n$. Therefore, the space complexity is bounded by $O(n)$.

To search for m nearest neighbors of the query point, the algorithm just needs to be modified to have a list containing m CNNPs and to set the CNNB's radius to be the longest nearest neighbor distance among all CNNPs currently in the list.

2.3. The modified kd-tree algorithm with an angularly bounded search space. The difficulty of searching for MCPs with the classic kd-tree is that we cannot specify a direction, or a range of directions, to search for the nearest neighbor.

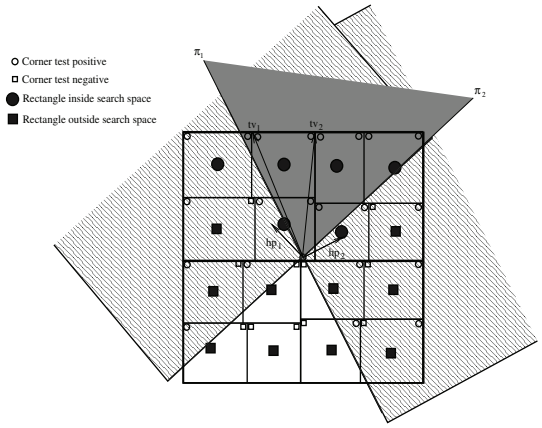


FIG. 2.1. *Illustration of the overlap test process of the modified kd-tree search algorithm. The lighter shades are two half spaces, and the darker shade is the intersection of the two half spaces, the area for the nearest neighbor search.*

This special feature of nearest neighbor search is critical to the implementation of an efficient MCP algorithm, as we have discussed in the previous section.

We propose to modify the kd-tree algorithm as follows.

The kd-tree will be constructed with the original algorithm. To search for the nearest neighbor in a bounded angular space, we pass to the search routine, in addition to a query point p, a pair of normal vectors hp_1 and hp_2, which specify the boundaries of the two half spaces. The intersection of the two half spaces represents the angular space to be searched. During a search, for each hyperrectangular subspace represented by a kd-tree node, we test if it overlaps with the bounded search space. If it does, we proceed the search as the classic kd-tree. Otherwise, we skip the subspace and go on to the next subspace.

To efficiently perform the above test, we maintain a pair of test vectors tv_1 and tv_2 that represent two out of 2^d vertices, of the d dimensional hyperrectangle being tested, such that vectors $tv_1 - p$ and $tv_2 - p$ have the greatest projections on the two normal vectors hp_1 and hp_2. At each node, we compute the projections of tv_1 and tv_2 on hp_1 and hp_2, respectively. If either projection is negative, then the hyperrectangle being tested does not overlap with the partial space specified by hp_1 and hp_2, the search space. If both projections are positive, then it is likely (but not always) that the hyperrectangle being tested overlaps with the search space. In other words, the above algorithm tests the necessary but not sufficient condition for overlap of the search space and a hyperrectangle. At the leaf level, we test each point within the leaf node and only update the CNNP and CNNB for the points that are inside the search space.

Figure 2.1 illustrates the above algorithm in a two-dimensional case. In the figure, the wedge-shaped area (in darker shade) is the intersection of two half spaces (in lighter shade) that needs to be searched; rectangular areas are the partitioned subspaces to be tested for overlap. Note that in addition to the 16 small rectangular subspaces, there are also 8, 4, and 2 larger rectangular subspaces in the figure that correspond to kd-tree nodes at different upper levels. The two upper vertices of each subspace are used as test vertices (vectors) tv_1 and tv_2, since they have the greatest projections on the normal vectors hp_1 and hp_2. Each circle marker indicates that the vertex of the subspace is tested positive and each square marker indicates a negative

result. A solid round marker in the center of the subspace indicates that the subspace is likely overlapped with the search space, and a solid square marker indicates that its subspace is outside the search area and should not be included for further examination. (For the clarity of the figure, we only plot the markers for the leaf level subspaces.)

The partitioned subspaces to be examined change as the search moves to a new node, while the vertices that the test vectors use for each subspace (the two upper vertices of each rectangle in the case shown in Figure 2.1) remains the same. These vertices are determined by the two normal vectors hp_1 and hp_2. For computational efficiency, we compute the tv_1 and tv_2 as we traverse the kd-tree. Due to the orthogonal partition scheme of the kd-tree, each time the search moves to a new node, there is at most one component of the test vector that needs to be updated.

The complete algorithms for the modified kd-tree is given here. The tree construction algorithm needs no modification. It is enclosed for the sake of completeness. The search algorithm contains several changes. The underlined statements indicate the additions and modifications to the classic kd-tree search algorithm.

2.4. A brief discussion of the performance of the modified kd-tree algorithm. First, we examine the situation of searching the nearest neighbor on the sphere without a pair of bounds. In this case, the modified kd-tree should behave just like the classic kd-tree.

Friedman, Bentley, and Finkel have given an expected search time analysis for the kd-tree algorithm [12]. They asserted that under certain assumptions, nearest neighbor search with kd-tree takes an $O(\log n)$ expected time. The two important assumptions are (1) the hyperrectangles are relatively compact, meaning that the maximum ratio between the longest and shortest sides are not too great, and (2) the dimensionality d of the search space is relatively small ($d \ll \log(n)$), and it is independent of the data size n. In our case, condition (2) is met perfectly, with n ranging from hundreds of thousands to millions and d being 3. As for condition (1), if we adopt the strategy used in [12] to partition the dimension with the maximum spread of data points in *dimToPartition*() (line 13 in Algorithm 2), we will have relatively compact hyperrectangles, thus the condition will be mostly satisfied as well.

Now we examine the nearest neighbor search in a lune. The computation at each node remains minimum, $O(d)$ operations. We compare the number of nodes visited by the modified kd-tree and the classic kd-tree. For the modified kd-tree, if it has the same CNNB radiuses along the traverse path as the classic kd-tree, because of the additional bound check for the lune, the search algorithm will visit less nodes (hyperrectangles). However, at the same time, because of this bound check, the search algorithm takes more time to find qualified CNNPs and to update the radius of the CNNB, causing more nodes to be visited. The situations could become extremely unfavorable for the modified kd-tree as the lunes to be searched become very narrow and the initial setting of the radius of the CNNB is significantly delayed. In these situations, it could improve the search performance dramatically if we can provide a reasonable initial value for the radius of the CNNB (search range) to start the search. Fortunately, for most applications including MCP search, it is not difficult to provide such an initial value. In MCP search, as we will find, the narrower the lune to be searched, the more accurate (smaller) search range is available to specify.

3. Search algorithm for MCPs on the sphere. We first define the spherical MCP for a given point p.

DEFINITION 3.1. *On the sphere $\mathcal{S}$, for a given point $p \in \mathcal{S}$ and a set of points $\mathcal{P} \in \mathcal{S}$, the spherical MCP for p (MCP(p)) of m vertices is the simple polygon on the*

ALGORITHM 1. KD-TREE SEARCH (MODIFIED FROM THE CLASSIC KD-TREE SEARCH ALGORITHM).

```
1: PROCEDURE SearchKDTree (p, hp_1, hp_2, inclusive, range, nni, nnd, m)
2: Set nni[1:m] (Indexes of nearest neighbors) to null
3: Set nnd[1:m] (Radiuses of the nearest neighbor balls) to range
4: Set dos[1:d] (Distances to other sides of the partition) to 0.0
5: Set the testing vertices tv_1[1:d] ← sign(hp_1[1:d]), tv_2[1:d] ← sign(hp_2[1:d])
6: {Make all arguments available in the global space accessible by all functions.}
7: Convert p = (φ, λ) to p = (x_1, x_2, x_3)
8: SearchTree(p, root)
9: END PROCEDURE
10: PROCEDURE SearchTree (query, node)
11: if node.leaf = true then
12:    For the points in the bucket within the search subspace, compute and update the
       nni[1:m] and nnd[1:m] if necessary.
13:    return
14: else
15:    d_s ← query(node.partDim ) - node.partVal
16:    if d_s < 0.0 then
17:       curPsum ← psum
18:       SearchTree(query, node.leftChild)
19:       psum ← curPsum + d_s^2 − dos[node.partDim]
20:       Save dos[node.partDim], tv_{1,2}[node.partDim]
21:       dos[node.partDim] ← d_s^2
22:       tv_{1,2}[node.partDim] ← max(node.partVal, tv_{1,2}[node.partDim])
23:       if psum < nnd[m] and InSearchSpace(tv_1, tv_2) then
24:          SearchTree(query, node.rightChild)
25:       end if
26:       Restore dos[node.partDim], tv_{1,2}[node.partDim] to the saved values.
27:    else
28:       curPsum ← psum
29:       SearchTree(query, node.rightChild)
30:       psum ← curPsum + d_s^2 − dos[node.partDim]
31:       Save dos[node.partDim], tv_{1,2}[node.partDim]
32:       dos[node.partDim] ← d_s^2
33:       tv_{1,2}[node.partDim] ← min(node.partVal, tv_{1,2}[node.partDim])
34:       if psum < nnd[m] and InSearchSpace(tv_1, tv_2) then
35:          SearchTree(query, node.leftChild)
36:       end if
37:       Restore dos[node.partDim], tv_{1,2}[node.partDim] to the saved values.
38:    end if
39: end if
40: return node
41: END PROCEDURE
42: FUNCTION InSearchSpace (tv_1, tv_2)
43: if (tv_1 − query) · hp_1 ≥ 0.0 ∧ (tv_2 − query) · hp_2 ≥ 0.0 ∧ inclusive = true then
44:    return true
45: end if
46: if (tv_1 − query) · hp_1 > 0.0 ∧ (tv_2 − query) · hp_2 > 0.0 ∧ inclusive = false then
47:    return true
48: end if
49: return false
50: END FUNCTION
```

ALGORITHM 2. KD-TREE CONSTRUCTION.

```
1: PROCEDURE Build-kd-tree
2: ml ← log n {maximum tree levels}
3: root ← BuildTree(points, 1, n, 1)
4: END PROCEDURE
5: FUNCTION BuildTree (points, lower, upper, level)
6: node ← newNode()
7: if level > ml then
8:    node.leaf ← true
9:    node.lowerBound ← lower, node.upperBound ← upper
10: else
11:   node.leaf ← false
12:   middle ← (lower + upper)/2
13:   node.partDim ← dimToPartition(points, l, u) {Find a dimension to partition}
14:   node.partVal ← points[node.partDim][middle] {Record partition value}
15:   partition(points, lower, upper, node.partDim) {Partition points into 2 subsets}
16:   node.leftChild ← BuildTree(points, lower, middle, level + 1)
17:   inode.rightChild ← BuildTree(points, middle, upper, level + 1)
18: end if
19: return node
20: END FUNCTION
```

sphere, defined by a sequence of m points $v_i \in \mathcal{P}$, such that (i) $\sum_{i=1}^{m} d(p, v_i)$ *is minimized, where* $d(.,.)$ *denotes the geodesic distance on* $\mathcal{S}$*, and* (ii) $p \in \Pi(SPL(p))$*, where* $\Pi(SPL(p))$ *is the point set of the spherical polygon defined by the vertex sequence* $\{v_1, v_2, \ldots, v_m\}$.

Apparently, according to Proposition 2.1, in the above definition $d(p, v_i)$ can be replaced with the Euclidean distance $\|p - v_i\|$.

Now we examine the relations and connections between the nearest neighbors of p and the vertices in MCP(p). Since it will be shown later that an MCP(p) ($m > 3$) can be easily constructed from the minimum covering triangle (MCT(p)), we start the discussion with the case of $m = 3$.

For the ease of description, we define the following notation.

Letters p, q, r, s, t, u, v, denote unit vectors on the sphere $\mathcal{S}$. As a convention for this article, p denotes the point for which the MCP is being searched, and r the nearest neighbor of p.

Geodesic circle $\pi_{u,v} = \{x \mid (u - v/(u \cdot v)) \cdot (x - u) = 0,\ x \in \mathcal{S}\}$.

Geodesic circle $\pi^{\perp}_{u,v} = \{x \mid (u \times v) \cdot (x - u) = 0,\ x \in \mathcal{S}\}$.

Half-sphere $\mathcal{H}_{u,v} = \{x \mid (u - v/(u \cdot v)) \cdot (x - u) \geq 0, x \in \mathcal{S}\}$.

Half-sphere $\mathcal{H}^{\perp}_{u,v} = \{x \mid (u \times v) \cdot (x - u) \geq 0, x \in \mathcal{S}\}$.

Bounded partial space on the sphere (Lune) $\mathcal{L}_{s,t,u,v} = \mathcal{H}^{\perp}_{s,t} \cap \mathcal{H}^{\perp}_{u,v}$

Here, $(u - v/(u \cdot v))$ is a vector in hyperplane spanned by u and v, tangential at u, from the projection of v on the tangent plane to u.

PROPOSITION 3.2. *Let r be the nearest neighbor of p, on the sphere $\mathcal{S}$. Then $r \in MCT(p)$.*

Proof. Suppose that MCT(p) $= \{v_1, v_2, v_3\}$ and r is the nearest neighbor of p. Without loss of generality, we assume that r and v_3 both belong to the partial space $\mathcal{L}_{p,v_2,v_1,p}$. Then $\{v_1, v_2, r\}$ also make a spherical triangle that covers p. Since r is closer to p than v_3, r should replace v_3 in MCT(p). ∎

Therefore, the search for the MCT(p) is reduced to finding a pair of two closest points to p, which together with r form a spherical triangle that covers p. Furthermore, if we can identify one of the two vertices, then the last one can be easily found by one nearest neighbor search in a specified lune.

It is apparent that the second vertex must be in the half-sphere $\mathcal{H}_{p,r}$. This half-sphere can be considered as a union of several lunes divided by geodesic circle $\pi^{\perp}_{p,q_i}$, where q_i are points in half-sphere $\mathcal{H}_{p,r}$. For each subspace (lune), we search for a closest pair of points (one point r_u in the lune and the other in a lune determined by p, r, and r_u) as the candidates for the MCT(p). The best pair of candidates, which has the smallest combined distance, is the pair of vertices that we are searching for.

The following proposition indicates how to find this pair of two vertices for a specified subspace after the first vertex of MCT(p), r, is found.

PROPOSITION 3.3. *Let r be the nearest neighbor of p. If r_u is the nearest neighbor of p in the partial space $\mathcal{L}_{q,p,r,p}$ ($\mathcal{L}_{p,q,p,r}$), where q is a point in half-sphere $\mathcal{H}_{p,r}$, then point r_u is the candidate for a member of MCT(p) for the partial space $\mathcal{L}_{q,p,p,r_u}$ ($\mathcal{L}_{p,q,r_u,p}$).*

Proof. We show the proposition for one of the two cases, as the two cases are entirely symmetric. Let r_u be the nearest neighbor of p in $\mathcal{L}_{q,p,r,p}$, and s be the nearest neighbor of p in $\mathcal{L}_{r_u,p,p,r}$. Apparently, r, r_u, and s makes a spherical triangle that covers p. Since no point in $\mathcal{L}_{q,p,p,r_u}$ is closer to p than r_u, and since point r'_u in $\mathcal{L}_{q,p,p,r_u}$ can only have an s' in $\mathcal{L}_{r'_u,p,p,r} \subset \mathcal{L}_{r_u,p,p,r}$ to form a spherical triangle that covers p, and by definition s' cannot be closer to p than s, the combined distance from r_u, s to p is minimum for any qualified triangle that has one vertex in the partial space $\mathcal{L}_{q,p,p,r_u}$. ∎

COROLLARY 3.4. *Let r be the nearest neighbor of p, and r_u be the nearest neighbor of p in $\mathcal{L}_{q_1,p,p,q_2}$, where q_1 and q_2 are points in $\mathcal{H}^{\perp}_{r,p} \cap \mathcal{H}_{p,r}$ and $\mathcal{H}^{\perp}_{p,r} \cap \mathcal{H}_{p,r}$, respectively. If $r_u \in \pi^{\perp}_{p,r}$, then r_u is the candidate for a member of MCT(p) for the partial space $\mathcal{L}_{q_1,p,p,q_2}$. If $q_1, q_2 \in \pi_{p,r}$ and $r_u \in \pi^{\perp}_{p,r}$, then $r_u \in$ MCT(p); in addition, if r_n is the nearest neighbor of p in $\mathcal{P} \setminus \{r, r_u\}$, then vertex set $\{r, r_u, r_n\}$ forms MCT(p).*

Proof. From Proposition 3.3, if $r_u \in \pi^{\perp}_{p,r}$, r_u is the candidate for a member of MCT(p) for the partial space $\mathcal{L}_{q_1,p,r,p} \cup \mathcal{L}_{p,r,p,q_2} = \mathcal{L}_{q_1,p,p,q_2}$. When $q_1, q_2 \in \pi_{p,r}$, $\mathcal{L}_{q_1,p,p,q_2} = \mathcal{H}_{p.r}$, the entire half-sphere. We know there is at least one vertex of MCT(p) in this half-sphere, thus $r_u \in$ MCT(p). Since r, p, and r_u lie on the same great circle, it is apparent that the nearest neighbor of p in $\mathcal{P} \setminus \{r, r_u\}$, r_n, together with r, r_u form a spherical triangle that covers p. ∎

With Propositions 3.2 and 3.3, using the modified nearest neighbor search algorithm, we can find the minimum covering spherical triangle for any given point p and a set of points $\mathcal{P}$ on the sphere with the following strategy. First, we find the nearest neighbor of p in $\mathcal{P}, r$, and add it to MCT(p) as the first vertex. Then we search the half-sphere $\mathcal{H}_{p,r}$ for the second vertex with this procedure.

Initially, we set hp_1, hp_2 to $\mathcal{H}_{p,r}$, and then repeat the following: finding the nearest neighbor r_{u_i} as a possible candidate for the second vertex in the lune defined by hp_1 and hp_2; (each geodesic bound of a lune is inclusive when it equals $\pi_{p,r}$, and exclusive otherwise) updating either hp_1 to $\mathcal{H}^{\perp}_{r_{u_i},p}$ or hp_2 to $\mathcal{H}^{\perp}_{p,r_{u_i}}$, depending on which side of the geodesic circle $\pi^{\perp}_{p,r}$ the point r_{u_i} resides; finding the nearest neighbor in the lune $\mathcal{L}_{r_{u_i},p,p,r}$ (or $\mathcal{L}_{p,r_{u_i},r,p}$), as a possible candidate for the third vertex; and updating the current best candidates for the second and third vertices of MCT(p) if necessary. To calculate the optimal nearest neighbor search ranges for the second and third vertices, one should use the distances from p to the current best candidates for the second and third vertices, r, and r_{u_i}.

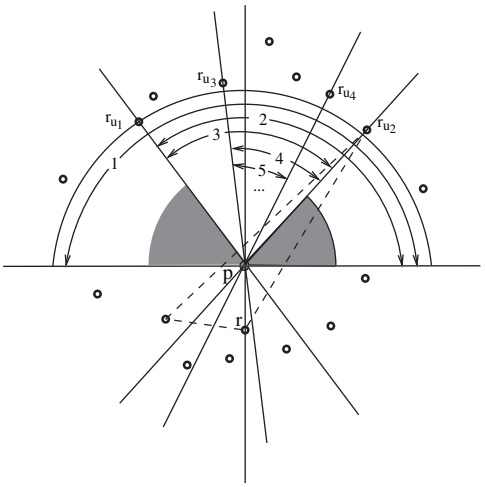


FIG. 3.1. *Illustration of the MCP search algorithm. In the figure, p is the query point, and the small circles are the data points; r is the nearest neighbor of p, and r_{ui} is the nearest neighbor found during the ith iteration, in the lune marked by the arc i.*

The above steps repeat until one of the following two conditions is satisfied: (1) there is no r_{u_i} that can be found in the current lune which satisfies $d(p, r_{u_i}) < d(p, v_1) + d(p, v_2) - d(p, r)$, where v_1 and v_2 are the current best candidates for the second and third vertices; (2) an r_{u_i} is found in hyperplane $\pi^{\perp}_{p,r}$ (Corollary 3.4).

Figure 3.1 illustrates the main idea of the MCT search algorithm with an example. The two shaded sectors indicate the lunes that have been searched after two iterations, where r_{u_1} is the nearest neighbor of p in $\mathcal{H}_{p,r}$. Each arc marked with i indicates the lune being searched in the ith iteration for the nearest neighbor r_{u_i}. The triangle in dashed lines is the MCT(p).

Once the minimum covering spherical triangle MCT(p) is found, it is straightforward to find the MCP(p) with more than three vertices.

PROPOSITION 3.5. *Let MCP(p) be a minimum covering spherical polygon of p with m vertices, where $m > 3$. Let v_1, v_2, and v_3 be the three vertices of the minimum covering spherical triangle, MCT(p). Let $\mathcal{P}$ be the set of points on the sphere to be searched. Then MCP(p) $= \{v_1, v_2, v_3\} \uplus \{v_4, \ldots, v_m\}$, where $\{v_4, \ldots, v_m\}$ is a sequence of $m-3$ nearest neighbors of p, in $\mathcal{P} \setminus \{v_1, v_2, v_3\}$, and operator $\uplus$ denotes the insert operation that creates a simple spherical polygon of m vertices.*

Proof. Since $\{v_4, \ldots, v_m\}$ are closer to p than any other points in $\mathcal{P} \setminus \{v_1, v_2, v_3\}$, $\sum_{i=1}^{m} d(p, v_i)$ is minimized. We just need to show MCP(p) $= \{\tilde{v_1}, \ldots, \tilde{v_m}\}$ covers p, where $\{\tilde{v_1}, \ldots, \tilde{v_m}\}$ is an appropriate permutation of the sequence $\{v_1, \ldots, v_m\}$. Following the same logic of the proof for Proposition 3.2, we infer that no member of $\{v_4, \ldots, v_m\}$ can be inside the spherical triangle MCT(p) $= \{v_1, v_2, v_3\}$, which implies that the spherical polygon created with $\{v_1, v_2, v_3\}$ and $\{v_4, \ldots, v_m\}$ can only be a union of the spherical triangle MCT(p) $= \{v_1, v_2, v_3\}$ and some additional spherical polygons that are formed with vertices $\{v_4, \ldots, v_m\}$ and vertices in MCT(p). Thus we can conclude that MCP(p) $= \{\tilde{v_1}, \ldots, \tilde{v_m}\}$ covers p. ∎

We give the entire algorithm in Algorithm 3. Note that in the description of the algorithm notation $\mathcal{H}_{u,v}$ and $\mathcal{H}^{\perp}_{u,v}$ denote the normal vectors of the associated half-spheres. Notice that the function *Insert()* at line 49 is not specified. There could be different ways and preferences to insert these $m-3$ nearest neighbor points to

ALGORITHM 3. MCP SEARCH ALGORITHM.

```
1: FUNCTION MCP-Search (p, m, mcp)
2: mcp[1 : m] ← 0, mcpd[1 : m] ← π, incl[1 : 2] ← true, nv ← 1
3: SearchKDTree (p, 0, 0, true, π, nni, nnd, 1)
4: mcp[1] ← nni[1], mcpd[1] ← nnd[1], r ← points[mcp[1]]        – Proposition 3.2
5: if p = r then
6:    CompleteMCP(mcp, nv)        – The MCP(p) will be formed with m NNs.
7:    return
8: end if
9: hpv_2[1] ← H_{p,r}, hpv_2[2] ← H_{p,r}, range ← π, nv ← 3
10: while true do
11:    SearchKDTree (p, hpv_2[1], hpv_2[2], incl, range, nni, nnd, 1)
12:    if nni = NULL then
13:       CompleteMCP(mcp, nv)     – The MCT(p) is found, to complete the MCP(p).
14:       return
15:    end if
16:    cvi[2] ← nni[1], cvd[2] ← nnd[1], r_u ← points[cvi[2]]
17:    if H^⊥_{r,p} · r_u = 0 then
18:       SearchKDTree (p, 0, 0, true, π, nni, nnd, 3)        – Corollary 3.4
19:       if nni[2] ≠ cvi[2] then
20:          cvi[3] ← nni[2], cvd[3] ← nnd[2]
21:       else
22:          cvi[3] ← nni[3], cvd[3] ← nnd[3]
23:       end if
24:       if cvd[2] + cvd[3] < mcpd[2] + mcpd[3] then
25:          mcpd[2 : 3] ← cvd[2 : 3], mcp[2 : 3] ← cvi[2 : 3]
26:       end if
27:       CompleteMCP(mcp, nv)     – The MCT(p) is found, to complete the MCP(p).
28:       return
29:    else
30:       if H^⊥_{r,p} · r_u > 0 then
31:          hpv_3[1] ← H^⊥_{r_u,p}, hpv_3[2] ← H^⊥_{p,r}, hpv_2[1] ← hpv_3[1], incl[1] ← false
32:       else
33:          hpv_3[1] ← H^⊥_{p,r_u}, hpv_3[2] ← H^⊥_{r,p}, hpv_2[2] ← hpv_3[1], incl[2] ← false
34:       end if
35:    end if
36:    range ← mcpd[2] + mcpd[3] − cvd[2]
37:    SearchKDTree (p, hpv_3[1], hpv_3[2], true, range, nni, nnd, 1)
38:    cvi[3] ← nni[1], cvd[3] ← nnd[1]
39:    if cvd[2] + cvd[3] < mcpd[2] + mcpd[3] then
40:       mcpd[2 : 3] ← cvd[2 : 3], mcp[2 : 3] ← cvi[2 : 3]
41:    end if
42:    range ← mcpd[2] + mcpd[3] − mcpd[1]
43: end while
44: END PROCEDURE
45: FUNCTION CompleteMCP (mcp, nv)        – Proposition 3.5
46: SearchKDTree (p, 0, 0, ∞, nni, nnd, m)
47: for i = 2 → m do
48:    if nni[i] ≠ mcp[1] . . . and nni[i] ≠ mcp[nv] then
49:       Insert(mcp, nni[i])
50:    end if
51: end for
52: END FUNCTION
```

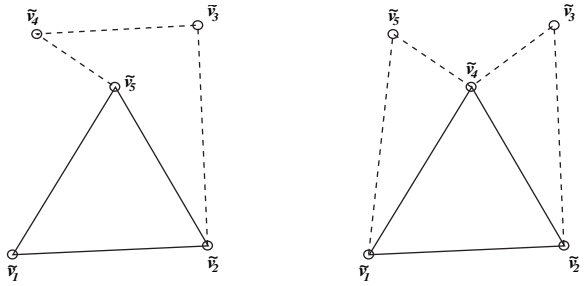


FIG. 3.2. *Illustration of two different ways to form a simple polygon from the same set of five vertices. In the figure, two identical MCTs are depicted with the solid lines, and two different* 5*-sided polygons are depicted with both solid and dashed lines, and two different sequences of vertices.*

an MCT(p) to create a simple polygon (Figure 3.2, for example). We leave the specifics of this function to the implementer. The upper bound for the time complexity of this insert function should be $O(m^2)$, which is independent of n, the size of $\mathcal{P}$.

3.1. Analysis of the time complexity of the MCP search algorithm. Through the discussion in section 2.4, we can assume that the modified kd-tree finds the nearest neighbor in an expected $O(\log(n))$ time. Now, we examine how many expected iterations there are for the while loop (from line 10 to line 43) in Algorithm 3.

For all practical purposes, we assume that the distribution of the data points over the sphere is largely uniform. With this assumption, we have the following claim.

CLAIM 3.6. *The expected number of iterations for the MCP search algorithm (Algorithm* 3*) is independent of the number of points that are uniformly distributed on the sphere.*

We argue the claim without a rigorous proof.

Let a set of n data points be randomly, uniformly distributed on the unit sphere, and let us assume that the number of points within a given distance from a random location follows Poisson distribution. Through some basic calculation of geometric probability, we find that the expected nearest neighbor distance for a random query point p, λ_d to be $1/2(4\pi/n)^{1/2}$. Furthermore, it can be calculated that the expected distance from a query point p to the nearest neighbor in half-sphere $\mathcal{H}_{p,r}$ is less than $2\lambda_d$.

Thus the expected search ranges for the second vertex, in the while loop iterations, should be less than $3\lambda_d$. On the other hand, within this search range, there could be, on expectation, no more than 10 data points, or, $O(1)$ number of data points.

Therefore, for uniformly distributed data points, the while loop in Algorithm 3 will terminate after a constant number (approximately 3–4, on expectation) of iterations.

The numerical experiment in next section confirms the claim, and it shows that the number of iterations for the while loop does not change when the number of data points varies from thousands to tens of millions. The number of iterations changes only when the distribution of the data points changes. It increases when the distribution becomes less uniform. The maximum number of iterations for the least uniform data distribution in the experiment is about 4.7.

With Claim 3.6, we can conclude that the expected time to perform an MCP search in a largely uniform data set is $O(\log n)$.

The above discussion also alludes to the worst case analysis of the MCP algorithm for unevenly distributed data sets. Suppose that within the search range there are

$O(n)$ data points and that these points are arranged in a way such that every point needs to be checked during the while loop; it is apparent that the worst case time complexity is $O(n \log n)$.

It is obvious that the worst case analysis is not applicable to most applications of the proposed MCP algorithm. For those applications in the numerical simulation of fluid dynamics on the sphere, the grid points will never be arranged like the contrived case, since any grid similar to the contrived grid layout will not be useful to discretize a dynamic system.

In some applications, it is sufficient to compute an approximate MCP. We can obtain an approximate algorithm from Algorithm 3 by simply changing the while loop block to a one-pass block. For this approximate MCP search algorithm, we have the same asymptotic expected time complexity. However, as is shown in the numerical experiment, on average, it will run twice as fast as the algorithm that finds the true MCP.

4. Numerical experiments. We carry out two numerical experiments: one for the nearest neighbor search on the sphere using the proposed modified kd-tree and the other for the MCP search on the sphere using the proposed MCP search algorithm. The second experiment is conducted for two versions of the MCP search algorithm that return a true MCP and an approximate MCP, respectively.

4.1. Experiment 1: Nearest neighbor search on the sphere using the modified kd-tree. We test the modified kd-tree algorithm with four types of randomly generated data sets. These data sets are generated with the following definitions, and they are shown in Figure 4.1.

Let X be a random variable following $U(0,1)$ distribution and Y be a random variable calculated from X. Let ϕ, λ be latitude and longitude values.

(a) Random data set with a uniform distribution on the sphere:

$$\mathcal{S}_{sp} = \{(\phi, \lambda) \mid \phi = \cos^{-1}(2X - 1.0) - \pi/2, \lambda = 2\pi X\}.$$

(b) Random data set with a uniform distribution in latitude and longitude:

$$\mathcal{S}_{\phi\lambda} = \{(\phi, \lambda) \mid \phi = (X - 0.5)\pi, \lambda = 2\pi X\}.$$

(c) Data set (a) plus a high density region $[\lambda_1, \lambda_2] \times [\phi_1, \phi_2]$:

$$\mathcal{S}_{sp \cup \{(\phi_1,\lambda_1),(\phi_2,\lambda_2)\}} = \mathcal{S}_{sp} \cup \mathcal{R}_{sp},$$

and

$$\mathcal{R}_{sp} = \{(\phi, \lambda) \mid \phi = \cos^{-1}(2Y - 1.0) - \pi/2, \lambda = \lambda_1(1.0 - X) + \lambda_2 X\},$$

where $Y = (1.0 - X)u_1 + Xu_2$ and $u_1 = (\cos(\phi_2 + \pi/2) + 1.0)/2.0, u_2 = (\cos(\phi_1 + \pi/2) + 1.0)/2.0$.

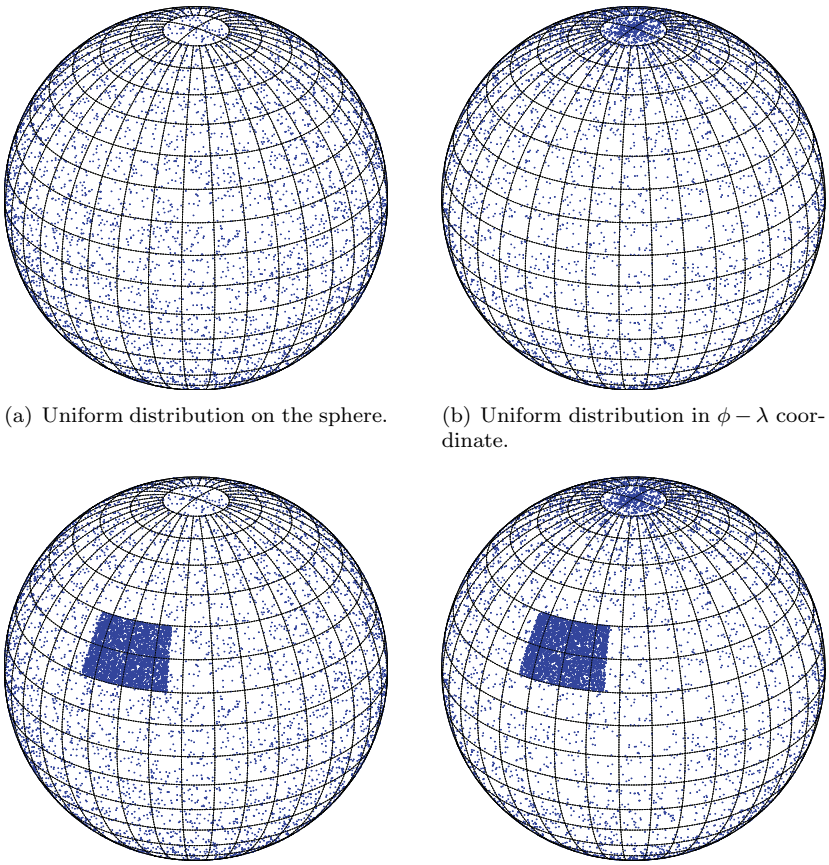
(d) Data set (b) plus a high density region $[\lambda_1, \lambda_2] \times [\phi_1, \phi_2]$:

$$\mathcal{S}_{\phi\lambda \cup \{(\phi_1,\lambda_1),(\phi_2,\lambda_2)\}} = \mathcal{S}_{\phi\lambda} \cup \mathcal{R}_{\phi\lambda},$$

and

$$\mathcal{R}_{\phi\lambda} = \{(\phi, \lambda) | \phi = \phi_1(1.0 - X) + \phi_2 X, \lambda = \lambda_1(1.0 - X) + \lambda_2 X\}.$$

Data set (a) simulates uniform spherical grids, such as icosahedral grids [5] and cubed-sphere grids [19], and data set (b) simulates uniform Cartesian grids, such as



(a) Uniform distribution on the sphere.

(b) Uniform distribution in $\phi - \lambda$ coordinate.

(c) Uniform distribution on the sphere with a high density region.

(d) Uniform distribution in $\phi - \lambda$ coordinate with a high density region.

FIG. 4.1. *Four types of randomly generated data sets used to test the performance of the modified kd-tree search algorithm and the MCP search algorithm.*

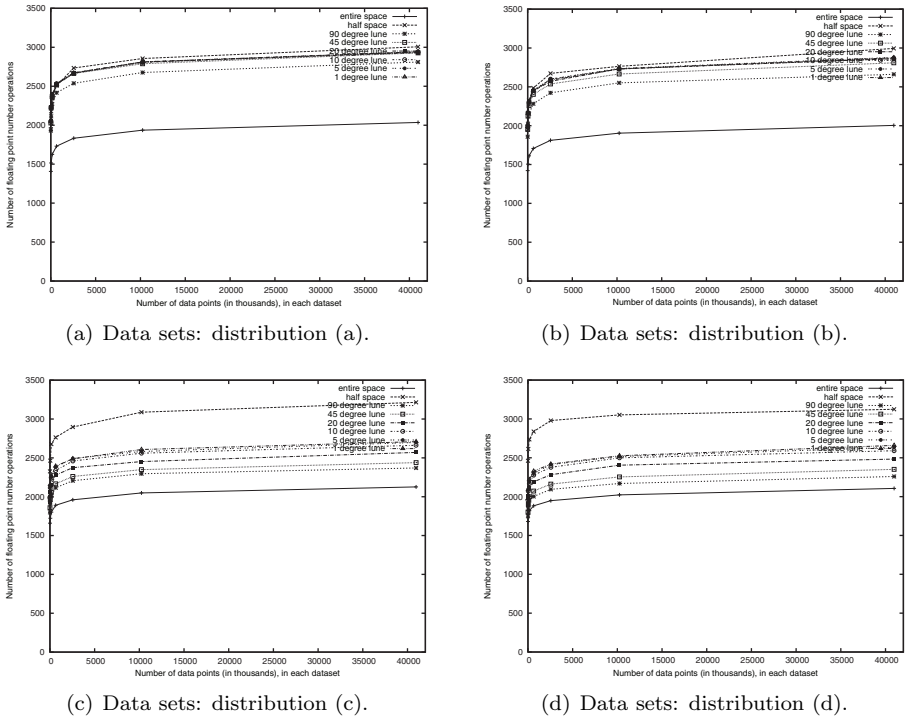
Gaussian grids [24]. Data sets (c) and (d) simulate those two types of grids with a locally nested or compressed high density region.

We conduct the nearest neighbor search test in several ways. First we test the full-space search, which is identical to the nearest neighbor search of the classic kd-tree. Thus the search algorithm of the modified kd-tree should perform similarly to that of the classic kd-tree. This test is to verify that the overhead is minimal due to the modifications.

Then we test the modified kd-tree for the half-sphere search, with half-spheres specified by randomly generated planes. For this test, we maintain the search range to be the length of the side of the entire search space. We expect to observe some performance differences compared to the full-space search.

Finally, we examine the performance of the modified kd-tree under the context of the MCP search. Pairs of planes are generated to form various sizes of lunes of random orientations, and we search the nearest neighbors within these lunes. In these tests, we set the search ranges to some conservative values which are multiples of the expected nearest neighbor distances of the data sets.

For each type of search, we test data sizes from about 10,000 to 40,000,000.



(a) Data sets: distribution (a).

(b) Data sets: distribution (b).

(c) Data sets: distribution (c).

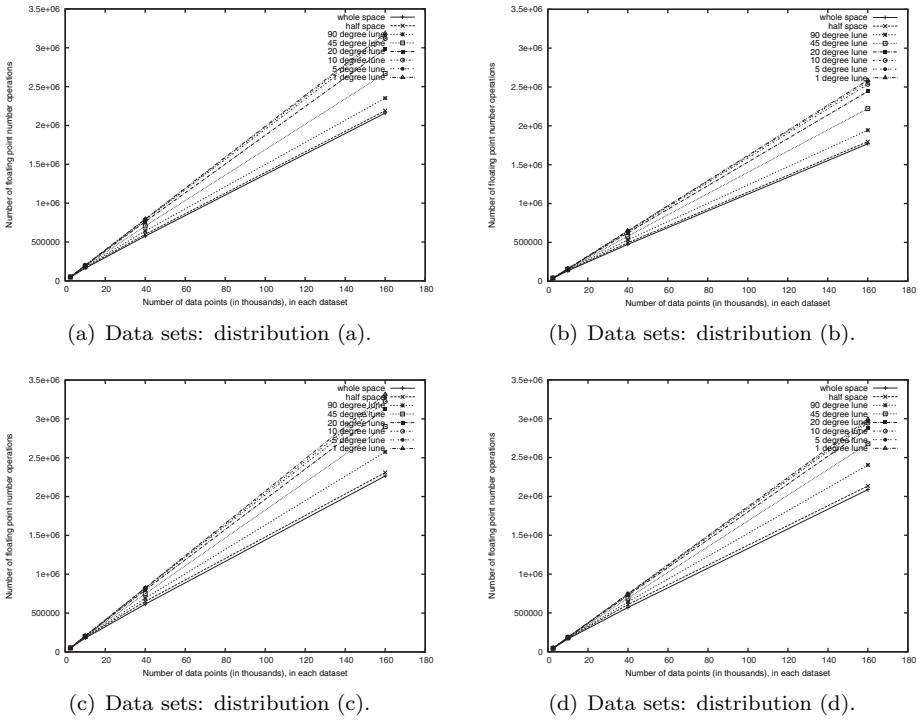
(d) Data sets: distribution (d).

FIG. 4.2. *Search test for the modified kd-tree algorithm. In each plot, the x axis indicates the number of grid points (in thousands), and y indicates the number of floating point number computations; each curve indicates the average number of floating point number computations for one nearest neighbor search.*

Figure 4.2 shows the test results. Each plot in the figure has its data sets created corresponding to one of the distributions depicted in Figure 4.1. Query points are drawn from the spherical uniform distribution (distribution (a)). A fixed number (163,842) of queries are made to the database for each test. An average number of floating point number operations per query for each test is obtained and used to plot a point in each curve. For each distribution, eight different sizes of lunes are searched for the nearest neighbor and eight curves are drawn in the corresponding plot. The angle sizes of the lunes are set to 360° (whole space search), 180° (half space search), $90°, 45°, 20°, 10°, 5°$, and 1°. For each lune which has size less than or equal to 90°, we set the search range to be five times the expected adjacent grid points distance.

We observe several things that are expected from the experiment. First, the numbers of floating point number operations used to find the nearest neighbor follow closely to the logarithm growth for all curves and in all four distributions of data points. Second, there is a small increase of computation for half sphere search, which is caused by the delay of updating the radius of the CNNB and by the "within the search space" bound test. We notice that with an appropriate initial search range, the performance of the search improves. Examining the plot closely, we also notice that for the same initial search range, the smaller the angular size of the lune, the more the computations. However, this increase of computation is rather minimum.

In addition, we implemented a nonhierarchical nearest neighbor search program. This program implements a linear search algorithm with some enhancements. At the initialization, grid points are sorted in their latitudes, and an appropriate number



(a) Data sets: distribution (a).

(b) Data sets: distribution (b).

(c) Data sets: distribution (c).

(d) Data sets: distribution (d).

FIG. 4.3. *Search test for enhanced nonhierarchical search algorithm. In each plot, the x, y axis and curves carry the same meaning as those in Figure* 4.2.

of latitudinal zones are created. Each latitudinal zone is pointed by an entry in an index table. When the algorithm searches the nearest neighbor of a query point, it just needs to search the data points in the same zone (of the query point) and two neighboring zones. Thus, the algorithm achieves a constant factor speed-up.

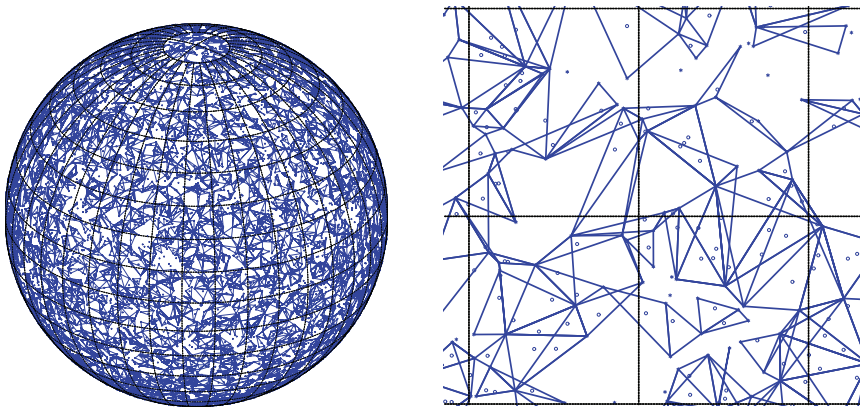
To compare the search performance, we run this nonhierarchical search algorithm in the same settings. The number of queries made to the data sets are also fixed to 163,842. The data sizes range from 2.56k to 160k in order to complete the test in a reasonable amount of time.

The test results in Figure 4.3 show a clear linear computational complexity in data size n for the nonhierarchical search algorithm. It should be noted that for some specific grids, the performance of the nonhierarchical algorithm can be further improved by introducing some special algorithms and data structures. However, these improvements are neither generally viable nor asymptotically significant.

4.2. Experiment 2: MCP search on the sphere. The MCP search algorithm finds a spherical covering polygon whose vertices are closest to the given point in their combined distance (Definition 3.1). Figure 4.4 shows an example of such a mesh of polygons (MCPs, $m = 3$).

In addition to the four data sets that we used to test the modified kd-tree in section 4.1, we also test two commonly used spherical grids in weather and climate modeling: icosahedral grid [5] and Gaussian grid [24].

An icosahedral grid is an unstructured grid created with recursive or nonrecursive subdivisions of the faces of an icosahedron and projections of the vertices to the sphere. Its grid points are rather uniformly and regularly distributed over the sphere. With



(a) (b)

FIG. 4.4. *A mesh of spherical MCTs. The triangles are created from a set of uniformly distributed random data points on the sphere (solid dots). The points to be covered (circular dots) by the triangles are obtained from the same distribution. In the figure,* (b) *is a zoomed-in area within* (a), *between* 20° *N and* 40° *N latitudes and* 10° *E and* 30° *E longitudes.*

the standard recursive construction algorithm, the upper bound of the maximum ratio of distances between adjacent grid points is about 1.195114 [23]. High resolution icosahedral grids have been used in several global models to simulate small scale atmospheric circulations [17, 21, 16].

A Gaussian grid is a Cartesian grid with its grid points equally spaced (in terms of their longitude values) along latitudinal circles, but these latitudinal circles are not equally spaced. Rather, they are located at the colatitudes of the arccosine of the roots of the Legendre polynomials, the abscissas of the Gaussian quadrature, hence the name. This grid is specially designed to efficiently discretize dynamic systems over the sphere that are simulated using spectral methods [18, 4, 15].

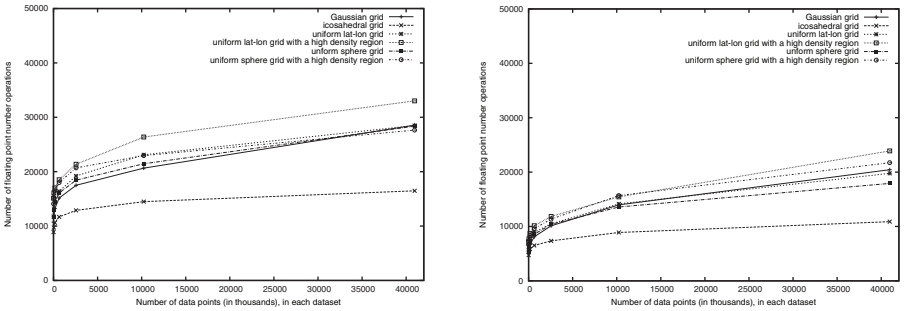
We test all six types of data sets of various data sizes, similar to what we have done for the modified kd-tree. Since we have already understood the performance of the nearest neighbor search algorithm of the modified kd-tree, and since we know it takes only one more call to the nearest neighbor search routine to complete a general MCP(p) ($m > 3$) after an MCT(p) is found (Proposition 3.5), we focus this test on how many iterations on average are required to find an MCT(p) in the while loop, comparing this number with the theoretical estimate. Table 4.1 shows the test results. The numbers in all rows are the average numbers of iterations in the while loop computed from 163,842 searches. All query points except for those in the last row are drawn from distribution (a). The query points in the last row are drawn from distribution (b).

It is clear that the number of iterations for the while loop in MCP search algorithm does not change with the data size, as predicted by the analysis. This number depends only on the distributions of the grid points and query points. The worst case, in our experiment, happened when both grid points and query points are nonuniformly distributed on the sphere with much higher density in the high latitude area.

Finally, we present the overall performances of the MCP search algorithm, in the same setting, in Figure 4.5. In the figure, plot (a) shows the performance of the MCP search described by Algorithm 3 and plot (b) shows the performance of the approximate version of the algorithm, which exits the while loop after one iteration. The figure shows pretty good search performance of both versions of the algorithm to

TABLE 4.1
MCP search experiment results.

Average number of iterations for the while loop in MCP search							
Grid Point #	10k	40k	160k	640k	2,560k	10,240k	40,960k
Gaussian grid	3.63	3.65	3.65	3.66	3.66	3.66	3.65
Icos. grid	3.11	3.11	3.11	3.11	3.11	3.11	3.11
Distribution(a)	3.49	3.50	3.49	3.49	3.49	3.49	3.48
Distribution(b)	3.48	3.49	3.49	3.49	3.50	3.49	3.48
Distribution(c)	3.50	3.49	3.50	3.49	3.49	3.49	3.48
Distribution(d)	3.51	3.50	3.49	3.49	3.49	3.48	3.48
Gauss. & Dist.(b)	4.44	4.54	4.60	4.66	4.70	4.69	4.27



(a) (b)

FIG. 4.5. *The MCP search performances. In both plots, the x axis indicates the number of grid points (in thousands), and y indicates the number of floating point number computations. Plot* (a): *exact MCP search. Plot* (b): *approximate MCP search.*

very large data size. We make two observations about the MCP search performance depicted in the figure: (1) for the most uniform grid, icosahedral grid, we get the best performance, which clearly demonstrates the logarithm behavior; the less uniform the grid, the greater the number of floating number computations; (2) for the approximate version of the MCP algorithm, the speed-ups are not proportional to the reduction of the number of iterations, which is closely related to the number of modified kd-tree searches. The reason is that during the first iteration in the while loop the search range is set to a much bigger default value (π, line 9 in Algorithm 2), compared to the searches that follow in the while loop; therefore more nodes in the kd-tree are visited, and more computations are needed. The second observation is consistent with the performance of the modified kd-tree in Figure 4.2.

In practice, the performance and robustness of the MCP search algorithm can be further enhanced by adding some practical safeguards for the special cases that will cause the algorithm to search a large amount of data points due to the round-off errors. One such special case would be trying to find an MCP for a query point at the poles from a Cartesian grid (Gaussian grid, for example), where all $O(\sqrt{n})$ nearest neighbors are equal-distant to the query point. Without a safeguard, the round-off error may delay the exit of the while loop significantly, resulting in many unnecessary computations. To avoid this, we can check if the exit condition is within the range of round-off error and exit the while loop earlier if necessary.

5. Conclusion. The analysis and numerical experiments have shown that the modified kd-tree and the MCP search algorithms are efficient and robust for various

spherical grids and for the data points that are randomly generated and uniformly distributed on the sphere. Even for the most unevenly distributed data sets in the numerical experiments we conducted, the search algorithms perform quite well. The proposed MCP algorithm takes an $O(n)$ space and $O(n \log n)$ time to complete the initialization. After that, it takes an expected $O(\log n)$ time to find the MCP(p) for a given query point p. In practice, the proposed MCP search algorithm reduces the computation time to find all MCPs in high resolution grids (ranging from millions to hundreds of millions of grid points) from a few hours to less than one minute.

The modified kd-tree algorithm can be used in other applications which require nearest neighbor searches in partial Euclidean spaces (or partial spheres). For example, if we want to find a chain of data points in a narrow stripe on the sphere, we can apply the modified kd-tree algorithm repeatedly with the newly found nearest neighbor as the new query point and with search lune covering the narrow stripe to be searched. In pattern recognition and vector quantization, the modified nearest neighbor search algorithm could be used to find a closest match in a partial feature vector space. There could be various applications of the modified kd-tree algorithm in scientific computations.

The MCP search algorithm and the modified kd-tree algorithm proposed could be used in general multidimensional settings. Nothing in the description of both algorithms limits search spaces to the three-dimensional Euclidean space (two-dimensional sphere). In fact, all computations are described in the general d dimensional Euclidean space. The MCP algorithm proposed can be used to find a set of m vertices on the $(d-1)$-sphere for a query point p, such that their combined angular distance (as defined in d-vector space) to p is minimized and the polygon defined by the m found vertices covers (embeds) the query point p. The performances of both search algorithms will decrease with the growth of dimension d. However, as long as $d \ll \log n$, the proposed algorithms will still provide a significant computational advantage over linear search.

Some considerations for parallel implementation have been given to the MCP search algorithm. Since the algorithm has an expected logarithm search time, the speed-up of the search time is not the main goal for parallelization. The primary motivation for parallelization is to use high performance computers with distributed memory systems to reduce preprocessing time and space on each processor, so that we can solve this search problem on those computer systems for a much greater data size. It is straightforward to implement the algorithm in coarse-grained parallelism on a parallel computer with a distributed memory system. At each processor, a local kd-tree is constructed from the grid points assigned to the processor. A nearest neighbor query request is broadcasted to all P processors, and each processor conducts its own local search with the same global search algorithm. After all searches are completed, the local results are gathered from P processors, and the global nearest neighbor, which is needed by the MCP search algorithm, is obtained from the local results. To speed up the updating of the radiuses of CNNBs, a global minimum radius of CNNB could be gathered and broadcasted once all local searches reach their first leaf nodes.

Acknowledgments. The author thanks Drs. Yuanfu Xie and Yiming Wang for their discussions and suggestions for the manuscript, and Drs. Alexander E. MacDonald, Jin-Luen Lee, and Stan Benjamin for their support for the research. The author also thanks the anonymous reviewers for their constructive comments.

REFERENCES

[1] P. K. Agarwal and J. Erickson, *Geometric range searching and its relatives*, in Advances in Discrete and Computational Geometry 23, B. Chazelle, J. E. Goodman, and R. Pollack, eds., AMS, Providence, RI, 1998, p. 156.

[2] P. Agarwal and J. Matoušek, *Ray shooting and parametric search*, in Proceedings of the 24th ACM Symposium on the Theory of Computing, 1992, pp. 517–526.

[3] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu, *An optimal algorithm for approximate nearest neighbor searching*, in Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 573–582.

[4] A. Baede, M. Jarraud, and U. Cubasch, *Adiabatic Formulation and Organization of ECMWF's Spectral Model*, Technical report, European Centre for Medium Range Weather Forecasts, Reading, UK, 1979.

[5] J. R. Baumgardner and P. O. Frederickson, *Icosahedral discretization of the two-sphere*, SIAM J. Numer. Anal., 22 (1985), pp. 1107–1115.

[6] J. Bentley, B. Weide, and A. Yao, *Optimal expected-time algorithms for closest point problems*, ACM Trans. Math. Software, 6 (1980), pp. 563–580.

[7] J. Bentley, *Multidimensional binary search trees used for associative searching*, Comm. ACM, 18 (1975), pp. 509–517.

[8] S. Brin, *Near neighbor search in large metric spaces*, in Proceedings of the 21st International Conference on Very Large Data Bases, 1995, pp. 574–584.

[9] K. L. Clarkson, *A randomized algorithm for closest-point queries*, SIAM J. Comput., 17 (1988), pp. 830–847.

[10] K. L. Clarkson, *An algorithm for approximate closest-point queries*, in Proceedings of the 10th ACM Symposium on Computational Geometry, 1994, pp. 160–164.

[11] C. Feustel and L. Shapiro, *The nearest neighbor problem in an abstract metric space*, Pattern Recognition Letters, 2 (1982), pp. 125–128.

[12] J. H. Friedman, J. L. Bentley, and R. A. Finkel, *An algorithm for finding best matches in logarithmic expected time*, ACM Trans. Math. Software, 3 (1977), pp. 209–226.

[13] P. Indyk and R. Motwani, *Approximate nearest neighbors: Towards removing the curse of dimensionality*, in Proceedings of the 30th Annual ACM Symposium on Theory of Computing, 1998, pp. 604–613.

[14] P. W. Jones, *First and second-order conservative remapping schemes for grids in spherical coordinates*, Mon. Wea. Rev., 127 (1998), pp. 2204–2210.

[15] M. Kanamitsu, *Description of the NMC global data assimilation and forecast system*, Wea. Forecasting, 4 (1989), pp. 335–342.

[16] J.-L. Lee and A. E. MacDonald, *A finite-volume icosshedral shallow water model on local coordinate*, Mon. Wea. Rev., 137 (2009), pp. 1422–1437.

[17] D. Majewski, D. LierMann, P. Prohl, B. Ritter, M. Buchhold, T. Hanisch, G. Paul, and W. Wergen, *The operational global icosahedral-hexagonal gridpoint model GME: Description and high-resolution tests*, Mon. Wea. Rev., 130 (2002), pp. 319–338.

[18] S. A. Orszag, *Transform method for calculation of vector coupled sums: Application to the spectral form of the vorticity equation*, J. Atmos. Sci., 27 (1970), pp. 890–895.

[19] W. M. Putman and S.-J. Lin, *A finite-volume dynamical core on the cubed-sphere grid*, in Numerical Modeling of Space Plasma Flows: Astronum-2008, Astronomical Society of the Pacific Conference Series 406, 2009, pp. 268–276.

[20] R. Swinbank and J. Purser, *Fibonacci grids: A novel approach to global modelling*, Q.J.R. Meteorol. Soc., 117 (2006), pp. 1769–1793.

[21] H. Tomita, K. Goto, and M. Satoh, *A new approach to atmospheric general circulation model: Global cloud resolving model NICAM and its computational performance*, SIAM J. Sci. Comput., 30 (2008), pp. 2755–2776.

[22] H. Tomita, M. Satoh, and K. Goto, *A new dynamical framework of global nonhydrostatic model using the icosahedral grid*, Fluid Dyn. Res., 34 (2004), pp. 357–400.

[23] N. Wang and J. Lee, *Geometric properties of the icosahedral-hexagonal grid on the two-sphere*, SIAM J. Sci. Comput., 33 (2011), pp. 2536–2559.

[24] W. M. Washington and C. L. Parkinson, *An introduction to three-dimensional climate modeling*, University Science Books, Sausalito, CA, 2005.

[25] P. N. Yianilos, *Data structures and algorithms for nearest neighbor search in general metric spaces*, in Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 311–321.